Intel® Integrated Performance Primitives

Developer Guide

Intel Integrated Performance Primitives 2020

Notices and Disclaimers

Contents

Notices and Disclaimers

Introducing the Intel® Integrated Performance Primitives

Use the Intel® Integrated Performance Primitives (Intel® IPP) to improve performance of multimedia, enterprise data, embedded, communications, and scientific/technical applications. The primitives are a common interface for thousands of commonly used algorithms. Using these primitives enables you to automatically tune your application to many generations of processors without changes in your application.

Intel IPP library provides high performance implementations of signal, image, and data processing functions for several hardware/instruction set generations. Code written with Intel IPP automatically takes advantage of available CPU capabilities. This can provide tremendous development and maintenance savings. You can write programs with one optimized execution path, avoiding the alternative of multiple paths (Intel® Streaming SIMD Extensions 2, Supplemental Streaming SIMD Extensions 3, Intel® Advanced Vector Extensions , etc.) to achieve optimal performance across multiple generations of processors.

The goal of the Intel IPP software is to provide algorithmic building blocks with

- a simple "primitive" C interface and data structures to enhance usability and portability
- faster time-to-market
- scalability with Intel® hardware

Intel IPP library is available as part of Intel® Parallel Studio XE and Intel® System Studio. It is also provided as a standalone package under the Community Licensing Program.

What's New

This Developer Guide documents the Intel® Integrated Performance Primitives (Intel® IPP) 2020 release .

The following changes have been made to the document:

- Updated the Using Intel IPP Custom Library Tool section to include instructions for building a DLL with Intel IPP Cryptography functionality.

Additionally, minor updates have been made to fix inaccuracies in the document.

Getting Help and Support

If you did not register your Intel® software product during installation, please do so now at the Intel® Software Development Products Registration Center. Registration entitles you to free technical support, product updates, and upgrades for the duration of the support term.

For general information about Intel technical support, product updates, FAQs, tips and tricks and other support questions, please visit http://www.intel.com/software/products/support/ and the Intel IPP forum http://software.intel.com/en-us/forums/intel-integrated-performance-primitives/.

NOTE
If your distributor provides technical support for this product, please contact them rather than Intel.

Notational Conventions

The following font and symbols conventions are used in this document:

Italic	*Italic* is used for emphasis and also indicates document names in body text, for example: see *Intel IPP Developer Reference*.
`Monospace lowercase`	Indicates filenames, directory names, and pathnames.
`Monospace lowercase mixed with UPPERCASE`	Indicates commands and command-line options, for example: `ippvars.bat ia32`
`UPPERCASE MONOSPACE`	Indicates system variables, for example: `$PATH`.
`monospace italic`	Indicates a parameter in discussions, such as routine parameters, for example: *`pSrc`*; makefile parameters, for example: *`function_list`*. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example: *`<ipp directory>`*.
[items]	Square brackets indicate that the items enclosed in brackets are optional.
{ item | item }	Braces indicate that only one of the items listed between braces can be selected. A vertical bar (|) separates the items.

Getting Started with Intel® Integrated Performance Primitives 1

This chapter helps you start using Intel® Integrated Performance Primitives (Intel® IPP) by giving a quick overview of some fundamental concepts and showing how to build an Intel® IPP program.

Finding Intel® IPP on Your System

Intel® Integrated Performance Primitives (Intel® IPP) installs in the subdirectory referred to as `<ipp directory>` inside `<install_dir>`. By default, the `<install_dir>` is:

- On Windows* OS: `C:/Program files (x86)/IntelSWTools/compilers_and_libraries_<version>/<target_os>` (on certain systems, instead of `Program Files (x86)`, the directory name is `Program Files`)
- On Linux* OS and macOS*: `/opt/intel/compilers_and_libraries_<version>/<linux|mac>`

The tables below describe the structure of the high-level directories on:

- Windows* OS
- Linux* OS
- macOS*

Windows* OS:

Directory	Contents
Subdirectories of *<ipp directory>*	
`bin`	Batch files to set environmental variables in the user shell
`include`	Header files for the library functions
`lib/ia32`	Single-threaded static libraries for the IA-32 architecture
`lib/intel64`	Single-threaded static libraries for the Intel® 64 architecture
`lib/<arch>/tl/<threading_type>`, where `<arch>` is one of `{ia32, intel64}`, and `<threading_type>` is one of `{tbb, openmp}`	Threading Layer static and dynamic libraries
`examples`	Symbolic link to Intel IPP example files
`components`	Intel IPP interfaces and example files
`tools/custom_library_tool_python`	Command-line and GUI tool for building custom dynamic libraries
`tools/<arch>/perfsys`	Command-line tools for Intel IPP performance testing
`tools/<arch>/staticlib`	Header files for disabling the Intel IPP static dispatcher (linking with one CPU-optimized variant)

Directory	Contents
Subdirectories of *<install_dir>*	
`redist/ia32/ipp`	Single-threaded DLLs for applications running on processors with the IA-32 architecture
`redist/intel64/ipp`	Single-threaded DLLs for applications running on processors with the Intel® 64 architecture

Linux* OS:

Directory	Contents
`bin`	Batch files to set environmental variables in the user shell
`include`	Header files for the library functions
`lib/ia32`	Single-threaded static libraries for the IA-32 architecture
`lib/intel64`	Single-threaded static libraries for the Intel® 64 architecture
`lib/`*<arch>*`/tl/`*<threading_type>*, where *<arch>* is one of `{ia32, intel64}`, and *<threading_type>* is one of `{tbb, openmp}`	Threading Layer static and dynamic libraries
`lib/`*<arch>*`/nonpic`	Non-PIC single-threaded static libraries
`lib/<arch>_and`	Intel IPP libraries for Android*
`examples`	Symbolic link to Intel IPP example files
`components`	Intel IPP interfaces and example files
`tools/custom_library_tool_python`	Command-line and GUI tool for building custom dynamic libraries
`tools/`*<arch>*`/perfsys`	Command-line tools for Intel IPP performance testing
`tools/`*<arch>*`/staticlib`	Header files for disabling the Intel IPP static dispatcher (linking with one CPU-optimized variant)

macOS*:

Directory	Contents
Subdirectories of*<ipp directory>*	
`bin`	Scripts to set environmental variables in the user shell
`include`	Header files for the library functions
`lib`	Single-threaded static FAT libraries
`lib/threaded`	Multi-threaded static FAT libraries
`lib/tl/`*<threading_type>*, where *<threading_type>* is one of `{tbb, openmp}`	Threading Layer static and dynamic FAT libraries

Directory	Contents
`examples`	Symbolic link to Intel IPP example files
`components`	Intel IPP interfaces and example files
`tools/custom_library_tool_python`	Command-line and GUI tool for building custom dynamic libraries
`tools/`*`<arch>`*`/perfsys`	Command-line tools for Intel IPP performance testing
`tools/`*`<arch>`*`/staticlib`	Header files for disabling the Intel IPP static dispatcher (linking with one CPU-optimized variant)

See Also

Notational Conventions

Setting Environment Variables

When the installation of Intel IPP is complete, set the environment variables in the command shell using one of the script files in the `bin` subdirectory of the Intel IPP installation directory:

On Windows* OS:

`ippvars.bat`	for the IA-32 and Intel® 64 architectures.

On Linux* OS and macOS*:

Architecture	Shell	Script File
IA-32 and Intel® 64	C	`ippvars.csh`
IA-32 and Intel® 64	Bash	`ippvars.sh`

When using the `ippvars` script, you need to specify the architecture as a parameter. For example:

- `ippvars.bat ia32`

  sets the environment for Intel IPP to use the IA-32 architecture on Windows* OS.
- `. ippvars.sh intel64`

  sets the environment for Intel IPP to use the Intel® 64 architecture on Linux* OS and macOS*.

The scripts set the following environment variables:

Windows* OS	Linux* OS	macOS*	Purpose
`IPPROOT`	`IPPROOT`	`IPPROOT`	Point to the Intel IPP installation directory
`LIB`	n/a	n/a	Add the search path for the Intel IPP single-threaded libraries
`PATH`	`LD_LIBRARY_PATH`	`DYLD_LIBRARY_PATH`	Add the search path for the Intel IPP single-threaded DLLs
`INCLUDE`	n/a	n/a	Add the search path for the Intel IPP header files

Compiler Integration

Intel® C++ Compiler and Microsoft Visual Studio* compilers simplify developing with Intel® IPP.

On Windows* OS, a default installation of Intel® IPP installs integration plug-ins. These enable the option to configure your Microsoft Visual Studio* project for automatic linking with Intel IPP.

Intel® C++ Compiler also provides command-line parameters to set the link/include directories:

- On Windows* OS:

  `/Qipp-link` and `/Qipp`
- On Linux* OS and macOS*:

  `-ipp-link` and `-ipp`

See Also

Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP
Linking Your Application with Intel(R) IPP

Building Intel® IPP Applications

The code example below represents a short application to help you get started with Intel® IPP:

```
#include "ipp.h"
#include <stdio.h>
int main(int argc, char* argv[])
{
   const IppLibraryVersion *lib;
   IppStatus status;
   Ipp64u mask, emask;

   /* Init IPP library */
   ippInit();
   /* Get IPP library version info */
   lib = ippGetLibVersion();
   printf("%s %s\n", lib->Name, lib->Version);

   /* Get CPU features and features enabled with selected library level */
   status = ippGetCpuFeatures( &mask, 0 );
   if( ippStsNoErr == status ) {
      emask = ippGetEnabledCpuFeatures();
      printf("Features supported by CPU\tby IPP\n");
      printf("-----------------------------------------\n");
      printf("  ippCPUID_MMX        = ");
      printf("%c\t%c\t",( mask & ippCPUID_MMX ) ? 'Y':'N',( emask & ippCPUID_MMX ) ? 'Y':'N');
      printf("Intel(R) Architecture MMX technology supported\n");
      printf("  ippCPUID_SSE        = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSE ) ? 'Y':'N',( emask & ippCPUID_SSE ) ? 'Y':'N');
      printf("Intel(R) Streaming SIMD Extensions\n");
      printf("  ippCPUID_SSE2       = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSE2 ) ? 'Y':'N',( emask & ippCPUID_SSE2 ) ? 'Y':'N');
      printf("Intel(R) Streaming SIMD Extensions 2\n");
      printf("  ippCPUID_SSE3       = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSE3 ) ? 'Y':'N',( emask & ippCPUID_SSE3 ) ? 'Y':'N');
      printf("Intel(R) Streaming SIMD Extensions 3\n");
      printf("  ippCPUID_SSSE3      = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSSE3 ) ? 'Y':'N',( emask & ippCPUID_SSSE3 ) ?
'Y':'N');
      printf("Intel(R) Supplemental Streaming SIMD Extensions 3\n");
      printf("  ippCPUID_MOVBE      = ");
      printf("%c\t%c\t",( mask & ippCPUID_MOVBE ) ? 'Y':'N',( emask & ippCPUID_MOVBE ) ?
'Y':'N');
      printf("The processor supports MOVBE instruction\n");
      printf("  ippCPUID_SSE41      = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSE41 ) ? 'Y':'N',( emask & ippCPUID_SSE41 ) ?
'Y':'N');
      printf("Intel(R) Streaming SIMD Extensions 4.1\n");
      printf("  ippCPUID_SSE42      = ");
      printf("%c\t%c\t",( mask & ippCPUID_SSE42 ) ? 'Y':'N',( emask & ippCPUID_SSE42 ) ?
'Y':'N');
      printf("Intel(R) Streaming SIMD Extensions 4.2\n");
      printf("  ippCPUID_AVX        = ");
      printf("%c\t%c\t",( mask & ippCPUID_AVX ) ? 'Y':'N',( emask & ippCPUID_AVX ) ? 'Y':'N');
      printf("Intel(R) Advanced Vector Extensions instruction set\n");
      printf("  ippAVX_ENABLEDBYOS  = ");
      printf("%c\t%c\t",( mask & ippAVX_ENABLEDBYOS ) ? 'Y':'N',( emask & ippAVX_ENABLEDBYOS ) ?
'Y':'N');
      printf("The operating system supports Intel(R) AVX\n");
      printf("  ippCPUID_AES        = ");
      printf("%c\t%c\t",( mask & ippCPUID_AES ) ? 'Y':'N',( emask & ippCPUID_AES ) ? 'Y':'N');
      printf("Intel(R) AES instruction\n");
      printf("  ippCPUID_SHA        = ");
      printf("%c\t%c\t",( mask & ippCPUID_SHA ) ? 'Y':'N',( emask & ippCPUID_SHA ) ? 'Y':'N');
      printf("Intel(R) SHA new instructions\n");
      printf("  ippCPUID_CLMUL      = ");
      printf("%c\t%c\t",( mask & ippCPUID_CLMUL ) ? 'Y':'N',( emask & ippCPUID_CLMUL ) ?
'Y':'N');
      printf("PCLMULQDQ instruction\n");
      printf("  ippCPUID_RDRAND     = ");
      printf("%c\t%c\t",( mask & ippCPUID_RDRAND ) ? 'Y':'N',( emask & ippCPUID_RDRAND ) ?
'Y':'N');
      printf("Read Random Number instructions\n");
      printf("  ippCPUID_F16C       = ");
      printf("%c\t%c\t",( mask & ippCPUID_F16C ) ? 'Y':'N',( emask & ippCPUID_F16C ) ? 'Y':'N');
      printf("Float16 instructions\n");
      printf("  ippCPUID_AVX2       = ");
      printf("%c\t%c\t",( mask & ippCPUID_AVX2 ) ? 'Y':'N',( emask & ippCPUID_AVX2 ) ? 'Y':'N');
      printf("Intel(R) Advanced Vector Extensions 2 instruction set\n");
      printf("  ippCPUID_AVX512F    = ");
      printf("%c\t%c\t",( mask & ippCPUID_AVX512F ) ? 'Y':'N',( emask & ippCPUID_AVX512F ) ?
'Y':'N');
      printf("Intel(R) Advanced Vector Extensions 3.1 instruction set\n");
      printf("  ippCPUID_AVX512CD   = ");
      printf("%c\t%c\t",( mask & ippCPUID_AVX512CD ) ? 'Y':'N',( emask & ippCPUID_AVX512CD ) ?
'Y':'N');
      printf("Intel(R) Advanced Vector Extensions CD (Conflict Detection) instruction set\n");
      printf("  ippCPUID_AVX512ER   = ");
      printf("%c\t%c\t",( mask & ippCPUID_AVX512ER ) ? 'Y':'N',( emask & ippCPUID_AVX512ER ) ?
'Y':'N');
      printf("Intel(R) Advanced Vector Extensions ER instruction set\n");
      printf("  ippCPUID_ADCOX      = ");
      printf("%c\t%c\t",( mask & ippCPUID_ADCOX ) ? 'Y':'N',( emask & ippCPUID_ADCOX ) ?
'Y':'N');
      printf("ADCX and ADOX instructions\n");
      printf("  ippCPUID_RDSEED     = ");
      printf("%c\t%c\t",( mask & ippCPUID_RDSEED ) ? 'Y':'N',( emask & ippCPUID_RDSEED ) ?
'Y':'N');
```

```
      printf("The RDSEED instruction\n");
      printf("  ippCPUID_PREFETCHW  = ");
      printf("%c\t%c\t",( mask & ippCPUID_PREFETCHW ) ? 'Y':'N',( emask & ippCPUID_PREFETCHW ) ?
'Y':'N');
      printf("The PREFETCHW instruction\n");
      printf("  ippCPUID_KNC        = ");
      printf("%c\t%c\t",( mask & ippCPUID_KNC ) ? 'Y':'N',( emask & ippCPUID_KNC ) ? 'Y':'N');
      printf("Intel(R) Xeon Phi(TM) Coprocessor instruction set\n");
   }
   return 0;
}
```

This application consists of three sections:

1. Initialize the Intel IPP library. This stage is required to take advantage of full Intel IPP optimization. The `ippInit()` function detects the processor type and sets the dispatcher to use the processor-specific code of the Intel® IPP library corresponding to the instruction set capabilities available. If your application runs without `ippInit()`, the Intel IPP library is auto-initialized with the first call of the Intel IPP function from any domain that is different from ippCore.

   In certain debugging scenarios, it is helpful to force a specific implementation layer using `ippSetCpuFeatures()`, instead of the best as chosen by the dispatcher.
2. Get the library layer name and version. You can also get the version information using the `ippversion.h` file located in the `/include` directory.
3. Show the hardware optimizations used by the selected library layer and supported by CPU.

Building the First Example with Microsoft Visual Studio* Integration on Windows* OS

On Windows* OS, Intel IPP applications are significantly easier to build with Microsoft* Visual Studio*. To build the code example above, follow the steps:

1. Start Microsoft Visual Studio* and create an empty C++ project.
2. Add a new c file and paste the code into it.
3. Set the include directories and the linking model as described in Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP.
4. Compile and run the application.

If you did not install the integration plug-in, configure your Microsoft* Visual Studio* IDE to build Intel IPP applications following the instructions provided in Configuring the Microsoft Visual Studio* IDE to Link with Intel® IPP.

Building the First Example on Linux* OS

To build the code example above on Linux* OS, follow the steps:

1. Paste the code into the editor of your choice.
2. Make sure the compiler and Intel IPP variables are set in your shell. For information on how to set environment variables see Setting Environment Variables.
3. Compile with the following command: `icc ipptest.cpp -o ipptest -I $IPPROOT/include -L $IPPROOT/lib/<arch> -lippi -lipps -lippcore`. For more information about which Intel IPP libraries you need to link to, see Library Dependencies by Domain and Linking Options.
4. Run the application.

Building the First Example on macOS*

To build the code example above on macOS*, follow the steps:

1. Paste the code into the editor of your choice.
2. Make sure the compiler and Intel IPP variables are set in your shell. For information on how to set environment variables see Setting Environment Variables.

3. Compile with the following command: `icc ipptest.cpp -o ipptest -I $IPPROOT/include -L $IPPROOT/lib/ -lippi -lipps -lippcore`. For more information about which Intel IPP libraries you need to link to, see Library Dependencies by Domain and Linking Options.
4. Run the application.

See Also

Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP
Configuring the Microsoft Visual Studio* IDE to Link with Intel® IPP
Setting Environment Variables
Library Dependencies by Domain
Linking Options
Dispatching
Intel® IPP Examples Directory Structure

Using Intel® IPP Examples

This section provides information on Intel IPP examples directory structure and examples build system.

Intel® IPP Examples Directory Structure

The Intel IPP package includes code examples, located in the `components_and_examples_<os>.zip` archive at the `<ipp directory>/components/` subdirectory. You can also access the archive using the symbolic link located at `<ipp directory>/examples/`.The `examples_core` subdirectory inside the archive contains the following files and directories:

Directory	Contents
`common`	Common code files for all examples
`documentation`	Documentation for the Intel IPP examples (`ipp-examples.html`)
`ipp_custom_dispatcher`	Custom dispatcher usage example
`ipp_fft`	Fast Fourier transformation example
`ipp_morphology`	Morphological reconstruction example
`ipp_resize_mt`	Image resizing example
`ipp_thread`	Example of external threading of Intel IPP functions

NOTE
Intel® IPP samples are no longer in active development and available as a separate download.

See Also

Finding Intel® IPP on Your System

Building Intel® IPP Examples

For building instructions refer to `examples_core/documentation/ipp-examples.html` provided with the `/<ipp directory>/components/components_and_examples_<os>.zip` archive.

See Also

Intel® IPP Examples Directory Structure

Finding the Intel® IPP Documentation

You can find getting started instructions and a listing of all the available online documents with links in the `get_started.htm` file available in the following directory:

- On Windows* OS: `C:/Program files (x86)/IntelSWTools/documentation_<version>/<locale>/ipp/common` (on certain systems, instead of `Program Files (x86)`, the directory name is `Program Files`)
- On Linux* OS and macOS*: `/opt/intel/documentation_<version>/<locale>/ipp/common`

Additional documentation on the Intel IPP examples (`documentation/ipp-examples.html`) is available in the `components_and_examples_<os>.zip` archive at the `<ipp directory>/components/` subdirectory.

The Intel IPP forum and knowledge base can be useful locations to search for questions not answered by the documents above. Please see: http://software.intel.com/en-us/forums/intel-integrated-performance-primitives/ .

See Also

Finding Intel® IPP on Your System

Intel® Integrated Performance Primitives Theory of Operation 2

This section discusses dispatching of the Intel® Integrated Performance Primitives (Intel® IPP) libraries to specific processors, provides functions and parameters naming conventions, and explains the data types on which Intel IPP performs operations. This section also provides Intel IPP domain details, including existing library dependencies by domain.

Dispatching

Intel® IPP uses multiple function implementations optimized for various CPUs. Dispatching refers to detection of your CPU and selecting the corresponding Intel IPP binary path. For example, the `ippie9` library in the `/redist/intel64/ipp` directory contains the image processing libraries optimized for 64-bit applications on processors with Intel® Advanced Vector Extensions (Intel® AVX) enabled such as the 2nd Generation Intel® Core™ processor family.

A single Intel IPP function, for example `ippsCopy_8u()`, may have many versions, each one optimized to run on a specific Intel® processor with specific architecture, for example, the 64-bit version of this function optimized for the 2nd Generation Intel® Core™ processor is `e9_ippsCopy_8u()`, and version optimized for 64-bit applications on processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE 4.2) is `y8_ippsCopy_8u()`. This means that a prefix before the function name determines CPU model. However, during normal operation the dispatcher determines the best version and you can call a generic function (`ippsCopy_8u` in this example).

Intel® IPP is designed to support application development on various Intel® architectures. This means that the API definition is common for all processors, while the underlying function implementation takes into account the strengths of each hardware generation.

By providing a single cross-architecture API, Intel IPP enables you to port features across Intel® processor-based desktop, server, and mobile platforms. You can use your code developed for one processor architecture for many processor generations.

The following table shows processor-specific codes that Intel IPP uses:

Description of Codes Associated with Processor-Specific Libraries

IA-32 Intel® architecture	**Intel® 64 architecture**	**Windows***	**Linux* OS**	**macOS***	**Description**
w7		+	+		Optimized for processors with Intel SSE2
	m7	+	+		Optimized for processors with Intel SSE3
s8	**n8**	+	+		Optimized for processors with Supplemental Streaming SIMD Extensions 3 (SSSE3)
p8	**y8**	+	+	+	Optimized for processors with Intel SSE4.2
g9	**e9**	+	+	+	Optimized for processors with Intel® Advanced Vector Extensions (Intel® AVX) and Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI)
h9	**l9**	+	+	+	Optimized for processors with Intel® Advanced Vector Extensions 2 (Intel® AVX2)
	n0	+	+		Optimized for 2nd Generation Intel® Xeon Phi™ Processor

IA-32 Intel® architecture	Intel® 64 architecture	Windows*	Linux* OS	macOS*	Description
	k0	+	+	+	Optimized for processors with Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Function Naming Conventions

Intel IPP functions have the same naming conventions for all domains.

Function names in Intel IPP have the following general format:

`ipp<data-domain><name>_<datatype>[_<descriptor>](<parameters>)`

NOTE
The core functions in Intel IPP do not need an input data type. These functions have `ipp` as a prefix without the data-domain field. For example, `ippGetStatusString`.

See Also
Core and Support Functions

Data-domain

The *data-domain* element is a single character indicating type of input data. Intel IPP supports the following data-domains:

`s`	one-dimensional operations on signals, vectors, buffers
`i`	two-dimensional operations on images, video frames

Primitive vs. Variant Name

The *name* element identifies the algorithm or operation of the function. The low-level algorithm that function implements is a *primitive*. This algorithm often has several *variants* for different data types and implementation variations.

For example, the `CToC` modifier in the `ippsFFTInv_CToC_32fc` function signifies that the inverse fast Fourier transform operates on complex floating point data, performing the complex-to-complex (CToC) transform.

Data Types

The *datatype* element indicates data types used by the function, in the following format:

`<bit depth><bit interpretation>`,

where

`bit depth = <1|8|16|32|64>`

and

`bit interpretation<`*`u|s|f`*`>[`*`c`*`]`

Here *u* indicates "unsigned integer", *s* indicates "signed integer", *f* indicates "floating point", and *c* indicates "complex".

For functions that operate on a single data type, the *datatype* element contains only one value.

If a function operates on source and destination signals that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

`<datatype> = <src1Datatype>[src2Datatype][dstDatatype]`

For more information about supported data types see the *Intel® IPP Reference Manual* available in the Intel® Software Documentation Library.

See Also

Intel® Software Documentation Library

Descriptor

The optional *descriptor* element describes the data associated with the operation. Descriptors are individual characters that indicate additional details of the operation.

The Intel IPP functions use the following descriptors:

Descriptor	Description	Example
`A`	Image data contains an alpha channel as the last channel, requires `C4`, alpha-channel is not processed.	`ippiFilterMax_8u_AC4R`
`A`*xx*	Advanced arithmetic operations with *xx* bits of accuracy.	`ippsPowx_32f_A11`
`C`	The function operates on a specified channel of interest (COI) for each source image.	`ippiSet_8u_C3CR`
`C`*n*	Image data consists of *n* channels. Possible values for *n*: 1, 2, 3, 4.	`ippiFilterBorder_32f_C1R`
`D`*x*	Signal is x-dimensional (default is `D1`).	`ippsConcat_8u_D2`
D		
`I`	Operation is in-place (default is not-in-place).	`ippsAdd_16s_I`
`L`	Platform-aware functions (see Using Intel IPP Platform-Aware Functions for details).	`ippiAdd_8u_C1RSfs_L`
`TL`	Threading layer functions (see Using Intel IPP Threading Layer Functions for details).	`ippiAdd_8u_C1RSfs_LT`
`M`	Operation uses a mask to determine pixels to be processed.	`ippiCopy_8u_C1MR`

Descriptor	Description	Example
`P`*n*	Image data consists of *n* discrete planar (not-interleaved) channels with a separate pointer to each plane. Possible values for *n*: 1, 2, 3, 4.	`ippiAlphaPremul_8u_AP4R`
`R`	Function operates on a defined region of interest (ROI) for each source image.	`ippiMean_8u_C4R`
`s`	Saturation and no scaling (default).	`ippiConvert_16s16u_C1Rs`
`Sfs`	Saturation and fixed scaling mode (default is saturation and no scaling).	`ippsConvert_16s8s_Sfs`

The descriptors in function names are presented in the function name in alphabetical order.

Some data descriptors are default for certain operations and not added to the function names. For example, the image processing functions always operate on a two-dimensional image and saturate the results without scaling them. In these cases, the implied descriptors `D2` (two-dimensional signal) and `s` (saturation and no scaling) are not included in the function name.

Parameters

The *`parameters`* element specifies the function parameters (arguments).

The order of parameters is as follows:

- All source operands. Constants follow vectors.
- All destination operands. Constants follow vectors.
- Other, operation-specific parameters.

A parameter name has the following conventions:

- All parameters defined as pointers start with *`p`*, for example, *`pPhase`*, *`pSrc`*; parameters defined as double pointers start with *`pp`*, for example, *`ppState`*. All parameters defined as values start with a lowercase letter, for example, *`val`*, *`src`*, *`srcLen`*.
- Each new part of a parameter name starts with an uppercase character, without underscore; for example, *`pSrc`*, *`lenSrc`*, *`pDlyLine`*.
- Each parameter name specifies its functionality. Source parameters are named *`pSrc`* or *`src`*, in some cases followed by names or numbers, for example, *`pSrc2`*, *`srcLen`*. Output parameters are named *`pDst`* or *`dst`* followed by names or numbers, for example, *`pDst2`*, *`dstLen`*. For in-place operations, the input/output parameter contains the name *`pSrcDst`* or *`srcDst`*.

Intel® Integrated Performance Primitives Domain Details

Intel IPP is divided into groups of related functions. Each subdivision is called *domain*, and has its own header file, static libraries, dynamic libraries, and tests. The table below lists each domain's code, header and functional area.

The file `ipp.h` includes Intel IPP header files with the exception of cryptography and generated functions. If you do not use cryptography and generated functions, include `ipp.h` in your application for forward compatibility. If you want to use cryptography functions, you must directly include `ippcp.h` in your application.

* available only within the Intel® System Studio suite

Library Dependencies by Domain

When you link to a certain Intel® IPP domain library, you must also link to the libraries on which it depends. The following table lists library dependencies by domain.

Library Dependencies by Domain

Domain	Domain Code	Depends on
Color Conversion	`CC`	`Core, VM, S, I`
String Operations	`CH`	`Core, VM, S`
Computer Vision	`CV`	`Core, VM, S, I`
Data Compression	`DC`	`Core, VM, S`
Image Processing	`I`	`Core, VM, S`
Signal Processing	`S`	`Core, VM`
Vector Math	`VM`	`Core`

To find which domain your function belongs to, refer to the *Intel® IPP Developer Reference* available in the Intel® Software Documentation Library.

See Also

Intel® Software Documentation Library

Linking Your Application with Intel® Integrated Performance Primitives 3

This section discusses linking options available in Intel® Integrated Performance Primitives (Intel® IPP).

The Intel IPP library supports the following linking options:

- Single-threaded dynamic
- Single-threaded static
- Multi-threaded dynamic
- Multi-threaded static
- Threading Layer static
- Threading Layer dynamic

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Linking Options

Intel® Integrated Performance Primitives (Intel® IPP) is distributed as:

- **Static library:** static linking results in a standalone executable
- **Dynamic/shared library:** dynamic linking defers function resolution until runtime and requires that you bundle the redistributable libraries with your application

The following table provides description of libraries available for linking.

	Single-threaded (non-threaded)	**Threading Layer** (externally threaded)
Description	Suitable for application-level threading	Implementation of application-level threading depends on single-threaded libraries
Found in	Main package After installation: `<ipp directory>/lib/<arch>` (static) and `<install_dir>//redist/<arch>/ipp`(dynamic)	Main package After installation: `<ipp directory>/lib/<arch>/tl/<threading_type>` , where `<threading_type>` is one of `{tbb, openmp}`
Static linking	Windows* OS: `mt` suffix in a library name (`ipp<domain>st.lib`)	Windows* OS: `mt` suffix in a library name (`ipp<domain>mt_tl.lib`)

	Linux* OS and macOS*: no suffix in a library name (`libipp<domain>.a`)	Linux* OS and macOS*: no suffix in a library name (`libipp<domain>_tl.a`) + single-threaded libraries dependency
Dynamic Linking	Default (no suffix) Windows* OS: `ipp<domain>.dll` Linux* OS: `libipp<domain>.so` macOS*: `libipp<domain>.dylib`	`_tl` suffix Windows* OS: `ipp<domain>_tl.dll` Linux* OS: `libipp<domain>_tl.so` macOS*: `libipp<domain>_tl.dylib` + single-threaded library dependency

To switch between Intel IPP libraries, set the path to the preferred library in system variables or in your project, for example:

- **Windows* OS**:

  Single-threaded: `SET LIB=<ipp directory>/lib/<arch>`

  Threading Layer: `SET LIB=<ipp directory>/lib/<arch>/tl/<threading_type>`. Additionally, set path to single-threaded libraries: `SET LIB=<ipp directory>/lib/<arch>`
- **Linux* OS/macOS***:

  Single-threaded: `gcc <options> -L <ipp directory>/lib/<arch>`

  Threading Layer: `gcc <options> -L <ipp directory>/lib/<arch>/tl<threading_type>`. Additionally, set path to single-threaded libraries: `gcc <options> -L <ipp directory>/lib/<arch>`

NOTE
On Linux* OS and macOS*, Intel IPP library depends on the following Intel® C++ Compiler runtime libraries: `libirc.a`, `libsvml.a`, and `libimf.a`. You should add a link to these libraries into your project. You can find these libraries in `<intel compiler directory>/lib` folders.

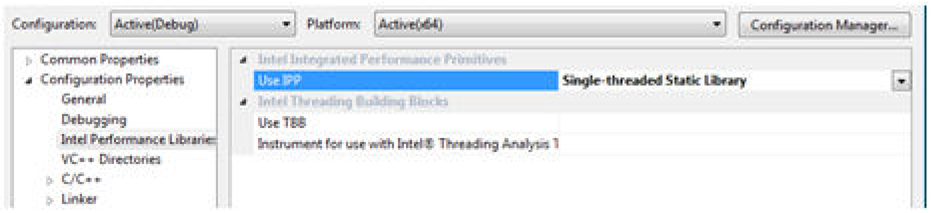
Threading Layer depends on the OpenMP* or Intel® Threading Building Blocks (Intel® TBB) library according to the selected threading type. You can find these libraries in `<intel compiler directory>/lib` or `<tbb directory>/lib` folders.

See Also
Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP
Configuring the Microsoft Visual Studio* IDE to Link with Intel® IPP
Library Dependencies by Domain

Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP

After a default installation of the Intel® IPP, you can easily configure your project to automatically link with Intel IPP. Configure your Microsoft* Visual Studio* project for automatic linking with Intel IPP as follows:

1. Go to **Project**>**Properties**>**Configuration Properties**>**Intel Performance Libraries**.
2. Change the **Use IPP** property setting by selecting one of the options to set the include directories and the linking model, as shown on the screen shot below.


Configuration: Active(Debug)
Platform: Active(x64)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
Intel Performance Libraries
VC++ Directories
C/C++
Linker
Intel Integrated Performance Primitives
Intel Threading Building Blocks
Use TBB
Instrument for use with Intel® Threading Analysis T

Use IPP: Single-threaded Static Library

Using Intel® Integrated Performance Primitives Platform-Aware Functions 4

Intel® Integrated Performance Primitives (Intel® IPP) library provides so-called platform-aware functions for signal and image processing. While the rest of Intel IPP functions support only signals or images of 32-bit integer size, Intel IPP platform-aware functions work with 64-bit object sizes if it is supported by the target platform.

The API of platform-aware functions is similar to the API of other Intel IPP functions and has only slight differences. You can distinguish Intel IPP platform-aware functions by the `L` suffix in the function name, for example, `ippiAdd_8u_C1RSfs_L`. With Intel IPP platform-aware functions you can overcome 32-bit size limitations.

Intel IPP platform-aware functions are declared in separate header files with the `_l` suffix, for example, `ippi_l.h`. However, you do not have to additionally include these headers in your application because they are already included in standard Intel IPP headers (without the `_l` suffix). Platform-aware functions cover only the functionality that is implemented in standard Intel IPP functions, and can be considered as additional flavors to the existing functions declared in standard Intel IPP headers.

Using Intel® Integrated Performance Primitives Threading Layer (TL) Functions 5

Intel® Integrated Performance Primitives (Intel® IPP) library provides threading layer (TL) functions for image processing. Intel IPP TL functions are visual examples of external threading for Intel IPP functions. Taking advantage of multithreaded execution and tile processing, Intel IPP TL functions enable you to overcome 32-bit size limitations.

TL functions are provided as:

- **Pre-built binaries:** header and library files have the `_tl` suffix and can be found in:
  - Header files: `<ipp directory>/include`
  - Library files: `<ipp directory>/lib/<arch>/tl/<threading_type>`, where *threading_type* is one of `{tbb, openmp}`.
- **Source code samples:** the source code and corresponding header files are available in the `components_and_examples_<os>.zip` archive inside the `<ipp directory>/components` subdirectory. For more information about the archive contents and source code building instructions, refer to Finding Intel® IPP TL Source Code Files and Building Intel® IPP TL Libraries from Source Code, respectively.

The API of TL functions is similar to the API of other Intel IPP functions and has only slight differences. You can distinguish Intel IPP TL functions by the `_LT` or `_T` suffix in the function name, for example, `ippiAdd_8u_C1RSfs_LT`. Intel IPP TL functions are implemented as wrappers over Intel IPP functions by using tiling and multithreading with OpenMP* or the Intel® Threading Building Blocks. For implementation details, please see the corresponding source code files.

NOTE
To use the Intel® IPP TL library on macOS*, you need to link your application with the Intel® OpenMP* `libiomp5` library, which is available at `<install_dir>/lib`.

See Also

Finding Intel® IPP TL Source Code Files
Building Intel® IPP TL Libraries from Source Code
Using Intel® Integrated Performance Primitives Platform-Aware Functions

Finding Intel® IPP TL Source Code Files

You can find the Intel IPP TL source code files in the `components_and_examples_<os>.zip` archive available in the `<ipp directory>/components` subdirectory. The library source code and header files are located in the `interfaces/tl` subdirectory.

Building Intel® IPP TL Libraries from Source Code

You can find the TL libraries source code and the `tl_resize` example in the `/components/interfaces/tl` directory inside the `components_and_examples_<os>` archive available in `<ipp directory>/components/`. Before building an application that uses TL, make sure that the `IPPROOT` environment variable is set correctly and points to the Intel IPP library location, for more information see Setting Environment Variables.

To build Intel IPP TL libraries and the `tl_resize` example, do the following:

Windows* OS

Prerequisites: The `tl_resize` example uses OpenGL rendering to display results. This requires Windows* SDK to be installed on your system. Usually Windows* SDK is provided with the Microsoft* Visual Studio* distribution. Alternatively, you can download Windows* SDK for your version of Windows* OS from https://www.microsoft.com. To disable the rendering part of `tl_resize`, remove the `ENABLE_RENDERING` macro from the preprocessors definitions.

1. Open the `tl.sln` file in Microsoft* Visual Studio*.
2. Choose the required configuration in the solution and build the solution using the **Build** command. The example will be linked with the newly built TL libraries from the same solution.

To build TL libraries on the Intel® Threading Building Blocks (Intel® TBB) library, you need to install the Intel TBB library (for the Intel IPP standalone package).

Linux* OS

Prerequisites: The `tl_resize` example uses OpenGL rendering to display results. This requires the following packages to be installed:

- `libx11-dev`
- `libgl1-mesa-dev`

Execute the following commands using gcc4 or higher:

- To build TL libraries:

```
make libs [ARCH=ia32|intel64] [CONF=release|debug] [TBBROOT=]
```

- To build the `tl_resize` example and TL libraries:

```
make all [ARCH=ia32|intel64] [CONF=release|debug] [RENDERER=0|1] [TBBROOT=]
```

If `TBBROOT` is set to the Intel® TBB installation root, TL libraries will be built with the TBB support. In this case, you need to install Intel TBB library (for the Intel IPP standalone package).

If `TBBROOT` is set to nothing, the OpenMP* support will be used.

macOS*

You can build TL libraries and the `tl_resize` example using the Apple Xcode* workspace provided in the `/components/interfaces/tl` directory. Alternatively, you can use makefiles:

- To build TL libraries:

```
make libs [ARCH=ia32|intel64] [CONF=release|debug] [TBBROOT=]
```

- To build the `tl_resize` example and TL libraries:

```
make all [ARCH=ia32|intel64] [CONF=release|debug] [RENDERER=0|1] [TBBROOT=]
```

If `TBBROOT` is set to the Intel® TBB installation root, TL libraries will be built with the TBB support. In this case, you need to install Intel TBB library (for the Intel IPP standalone package).

If `TBBROOT` is set to nothing, the OpenMP* support will be used.

To build TL libraries on OpenMP*, you need to use the custom compiler, because the macOS* Clang compiler does not support OpenMP*. You can do one of the following:

- Use Intel® Compiler (installed with Intel® Parallel Studio XE) with Makefile scripts:

  1. Change directory to the `/components/interfaces/tl` folder
  2. Run the following command:

```
make CC=icc CXX=icc
```

  3. Before running the application, specify the path to the OpenMP* library: `DYLD_LIBRARY_PATH=/opt/intel/compilers_and_libraries/mac/lib/`
- Use mainline Clang compiler that can work with Xcode*:

  1. Install Homebrew* from http://brew.sh/
  2. Run `brew install llvm`. The latest mainline `llvm` version will be installed to `/usr/local/opt/llvm` (if the installation folder is different, you will need to update Xcode* paths for **CC** and **Library Search Paths** properties).
  3. Build Xcode* projects for TL libraries.

See Also

Setting Environment Variables

Using Custom Library Tool for Intel® Integrated Performance Primitives 6

With the Intel® Integrated Performance Primitives (Intel® IPP) Custom Library Tool, you can build your own dynamic library containing only the Intel IPP/Intel IPP Cryptography functionality that is necessary for your application.

The use of custom libraries built with the Custom Library Tool provides the following advantages:

- **Package size**. Your package may have much smaller size if linked with a custom library because standard dynamic libraries additionally contain all optimized versions of Intel IPP/Intel IPP Cryptography functions and a dispatcher. The following table compares the contents and size of packages for an end-user application linked with a custom dynamic library and an application linked with the standard Intel IPP dynamic libraries:

Application linked with custom DLL	Application linked with Intel IPP dynamic libraries
`ipp_test_app.exe` (for Windows*) or `ipp_test_app` (for Linux* OS and macOS*)	`ipp_test_app.exe` (for Windows*) or `ipp_test_app` (for Linux* OS and macOS*)
`ipp_custom_{dll|so}.{dll|so|dylib}`	`ippi.{dll|so|dylib}`
	`ippig9.{dll|so|dylib}`
	`ippih9.{dll|so|dylib}`
	`ippip8.{dll|so|dylib}`
	`ippipx.{dll|so|dylib}`
	`ippis8.{dll|so|dylib}`
	`ippiw7.{dll|so|dylib}`
	`ipps.{dll|so|dylib}`
	`ippsg9.{dll|so|dylib}`
	`ippsh9.{dll|so|dylib}`
	`ippsp8.{dll|so|dylib}`
	`ippspx.{dll|so|dylib}`
	`ippss8.{dll|so|dylib}`
	`ippsw7.{dll|so|dylib}`
	`ippcore.{dll|so|dylib}`
Package size: 0.1 Mb	**Package size: 121.5 Mb**

- **Smooth transition to a higher version of Intel IPP/Intel IPP Cryptography**. You can easily build the same custom dynamic library from a higher version of Intel IPP/Intel IPP Cryptography and substitute the libraries in your application without relinking.

NOTE The current Python* version of the Intel IPP Custom Library Tool supports the host-host configuration only, the host-target configuration is currently not supported.

System Requirements for Custom Library Tool

Recommended hardware:

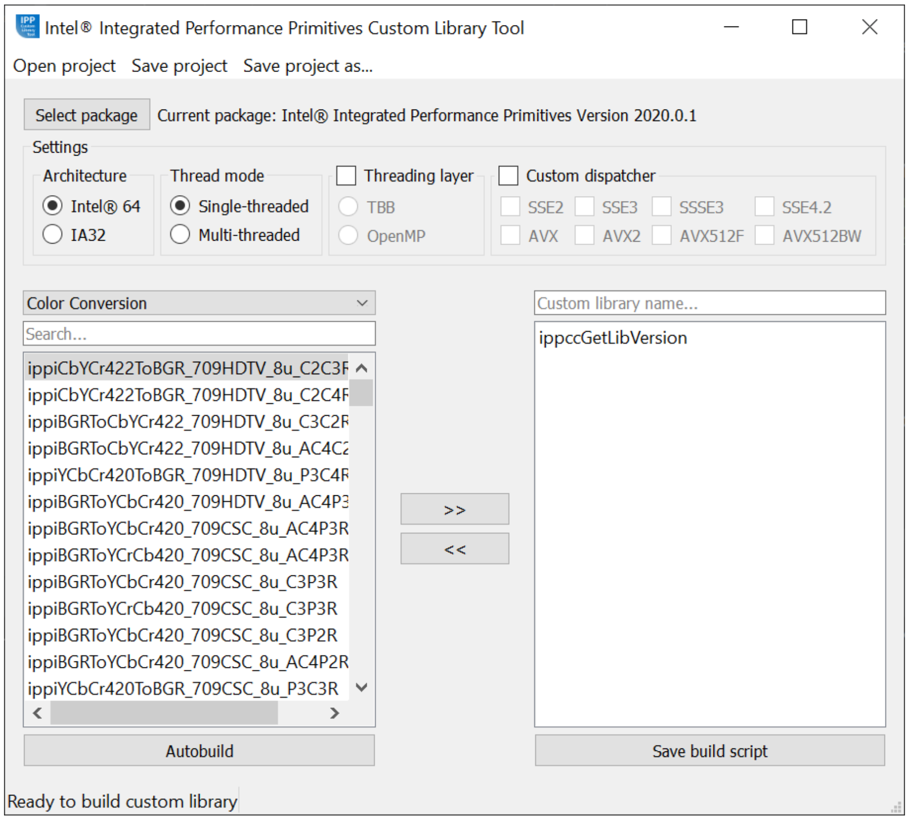
- System based on the 2nd Generation Intel® Core i3, i5, i7 or higher processor

Software requirements:

- Visual Studio* 2013 (or higher) Redistributable Packages
- Python* 3.7
- PyQt5 (required only for the GUI version of the Intel IPP Custom Library Tool)

Operation Modes

You can choose one of two tool operation modes, as shown at the screen shot below:



Current package: Intel® Integrated Performance Primitives Version 2020.0.1

Architecture: [✓ Intel® 64] IA32
Thread mode: [✓ Single-threaded] Multi-threaded
☐ Threading layer: TBB, OpenMP
☐ Custom dispatcher: ☐ SSE2 ☐ SSE3 ☐ SSSE3 ☐ SSE4.2 ☐ AVX ☐ AVX2 ☐ AVX512F ☐ AVX512BW

Color Conversion
Search...

ippiCbYCr422ToBGR_709HDTV_8u_C2C3R
ippiCbYCr422ToBGR_709HDTV_8u_C2C4R
ippiBGRToCbYCr422_709HDTV_8u_C3C2R
ippiBGRToCbYCr422_709HDTV_8u_AC4C2
ippiYCbCr420ToBGR_709HDTV_8u_P3C4R
ippiBGRToYCbCr420_709HDTV_8u_AC4P3
ippiBGRToYCbCr420_709CSC_8u_AC4P3R
ippiBGRToYCrCb420_709CSC_8u_AC4P3R
ippiBGRToYCbCr420_709CSC_8u_C3P3R
ippiBGRToYCrCb420_709CSC_8u_C3P3R
ippiBGRToYCbCr420_709CSC_8u_C3P2R
ippiBGRToYCbCr420_709CSC_8u_AC4P2R
ippiYCbCr420ToBGR_709CSC_8u_P3C3R

Custom library name...

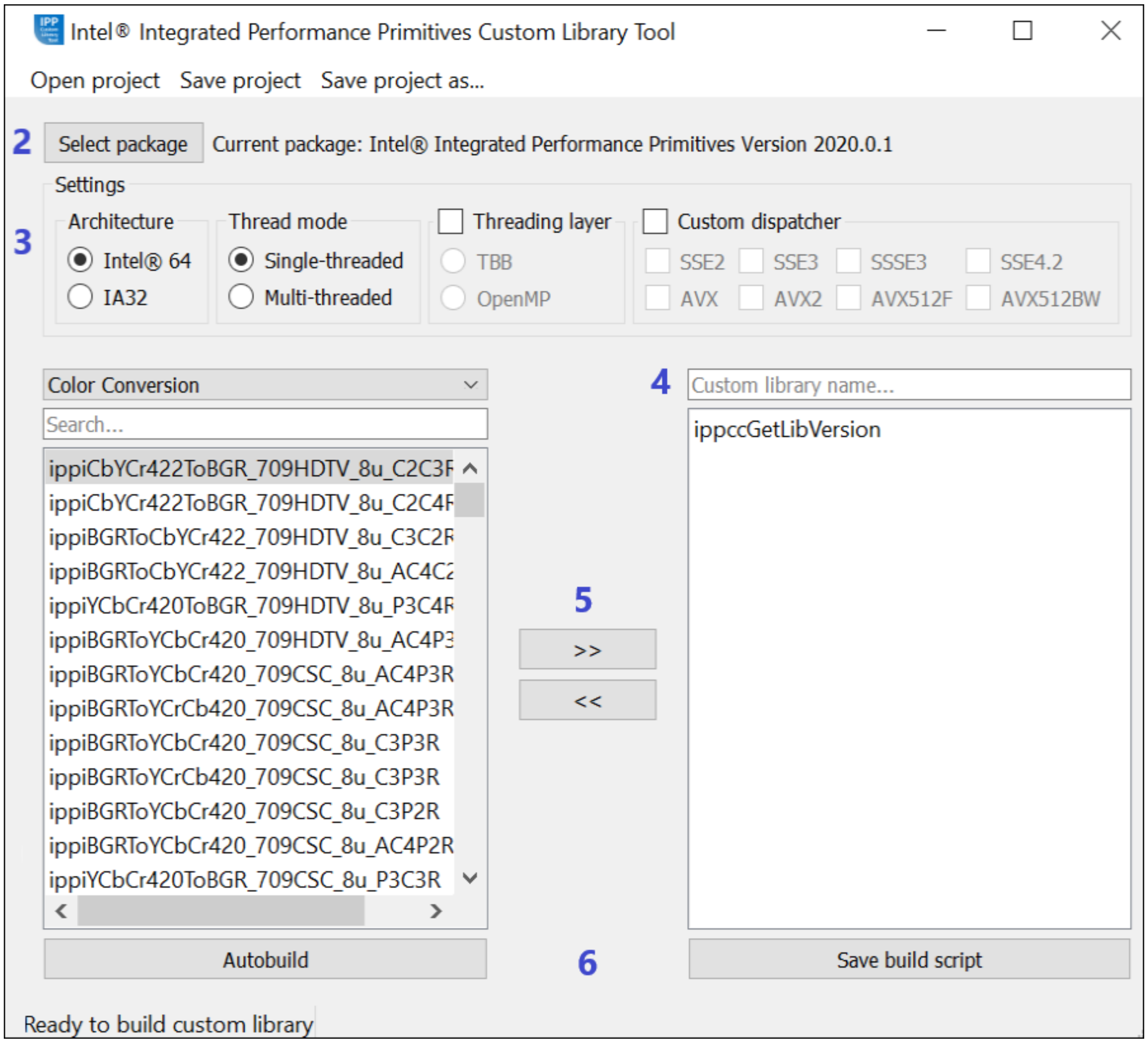
ippccGetLibVersion

Ready to build custom library

- **Auto build**. The tool automatically sets the environment and builds a dynamic library.
- **Save script**. The tool generates and saves a custom build script.

Building a Custom DLL with Custom Library Tool

Follow the steps below to build a custom dynamic library using the Intel IPP Custom Library Tool:


Intel® Integrated Performance Primitives Custom Library Tool
Open project Save project Save project as...
2
Select package
3
4
5
>>
<<
Autobuild
6
Save build script

Current package: Intel® Integrated Performance Primitives Version 2020.0.1

Settings

Architecture: [✓ Intel® 64] IA32

Thread mode: [✓ Single-threaded] Multi-threaded

Threading layer: TBB, OpenMP

Custom dispatcher: SSE2, SSE3, SSSE3, SSE4.2, AVX, AVX2, AVX512F, AVX512BW

Color Conversion

Search...

ippiCbYCr422ToBGR_709HDTV_8u_C2C3R
ippiCbYCr422ToBGR_709HDTV_8u_C2C4R
ippiBGRToCbYCr422_709HDTV_8u_C3C2R
ippiBGRToCbYCr422_709HDTV_8u_AC4C2
ippiYCbCr420ToBGR_709HDTV_8u_P3C4R
ippiBGRToYCbCr420_709HDTV_8u_AC4P3
ippiBGRToYCbCr420_709CSC_8u_AC4P3R
ippiBGRToYCrCb420_709CSC_8u_AC4P3R
ippiBGRToYCbCr420_709CSC_8u_C3P3R
ippiBGRToYCrCb420_709CSC_8u_C3P3R
ippiBGRToYCbCr420_709CSC_8u_C3P2R
ippiBGRToYCbCr420_709CSC_8u_AC4P2R
ippiYCbCr420ToBGR_709CSC_8u_P3C3R

Custom library name...

ippccGetLibVersion

Ready to build custom library

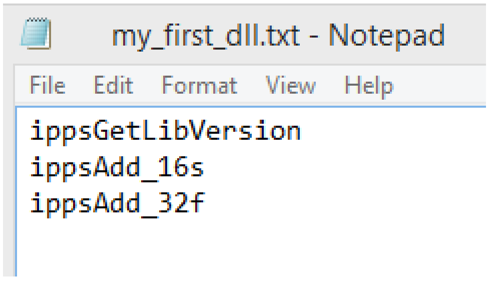
1. Run `python main.py` to launch the GUI version of the tool.
2. Select the Intel IPP or Intel IPP Cryptography package (optional). If you run the tool inside the Intel IPP or Intel IPP Cryptography package, the current one will be used as default. Otherwise, you need to provide the path to the package.
3. Configure your custom library.
4. Set the library name.
5. Select functions from the list. You can build a dynamic library containing Intel IPP or Intel IPP Cryptography functionality, but not both. If you need to add threaded functions to the custom list, select **Threading layer** checkbox to show the list of threaded functions.
6. Build the library automatically (if available) or save a build script.

NOTE
You can save the configuration and the list of custom functions as a project by clicking **Save project** or **Save project as...**. The project is saved as a file with the `.cltproj` extension. Then you can open this project by clicking **Open project** button.

Using Console Version of Custom Library Tool

Follow the steps below to build a custom dynamic library using console version of the Custom Library Tool:

1. Define a list of Intel IPP functions that the Intel IPP Custom Library Tools should export to your custom dynamic library. See the example text file below:



```
ippsGetLibVersion
ippsAdd_16s
ippsAdd_32f
```

2. Run `python main.py` with the following parameters:

`-c, --console`	Launches the console version of the tool (the GUI version is used by default).
`-cp, --cryptography`	Enables building of a DLL with Intel IPP Cryptography functionality.
`-g, --generate`	Enables the script generation mode (the build mode is used by default).
`-n <name>, --name <name>`	Output library name.
`-p <path>, --path <path>`	Path to the output directory.
`-root <root_path>`	Path to Intel IPP or Intel IPP Cryptography package root directory
`-f <function>, --function <function>`	Name of a function to be included into your custom dynamic library.
`-ff <functions_file>, --functions_file <functions_file>`	Path to a file with a list of functions to be included into your final dynamic library (the `-f` or `--function` flag can be used to add functions on the command line).
`-arch={ia32|intel64}`	Enables all actions for the IA-32 or the Intel ® 64 architecture (Intel ® 64 architecture is used by default).
`-mt, --multi-threaded`	Enables multi-threaded libraries (single-threaded libraries are used by default).
`-tl={tbb|openmp}`	Sets Intel TBB or OpenMP* as the threading layer.
`-d , --custom_dispatcher <cpu_set>`	Sets the exact list of CPUs that must be supported by custom dynamic library and generates a C-file with the custom dispatcher.
`-h, --help`	Prints command help.

For example:

```
# Generate build scripts in console mode
# with the output dynamic library name "my_custom_dll.dll"
# with functions defined in the "functions.txt" file
# optimized only for processors with
# Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
# using multi-threaded IA-32 Intel IPP libraries

python main.py -c -g
-n my_custom_dll
-p “C:\my_project”
-ff “C:\my_project\functions.txt”
-d avx512bw
-arch=ia32 -mt
```

Using Integration Wrappers for Intel® Integrated Performance Primitives

7

Intel® Integrated Performance Primitives (Intel® IPP) Integration Wrappers aggregate Intel IPP functionality in easy-to-use functions and help to reduce effort required to integrate Intel IPP into your code.

Integration Wrappers consist of C and C++ interfaces:

- **C interface** aggregates Intel IPP functions of similar functionality with various data types and channels into one function. Initialization steps required by several Intel IPP functions are implemented in one initialization function for each functionality. To reduce the size of your code and save time required for integration, the wrappers handle all memory management and Intel IPP function selection routines.
- **C++ interface** wraps around the C interface to provide default parameters, easily initialized objects as parameters, exception handling, and objects for complex Intel IPP functions with automatic memory management for specification structures.

In general, Integration Wrappers are designed to improve user experience with threading of Intel IPP functions and tiling.

Integration Wrappers are provided as a separate download. For more information about the main concepts, usage, and implementation details, refer to the *Developer Guide and Reference for Intel IPP Integration Wrappers* document available with the Integration Wrappers package.

Programming Considerations 8

Core and Support Functions

There are several general purpose functions that simplify using the library and report information on how it is working:

- `Init`/`GetCpuFeatures`/ `SetCpuFeatures`/`GetEnabledCpuFeatures`
- `GetStatusString`
- `GetLibVersion`
- `Malloc`/`Free`

Init/GetCpuFeatures/ SetCpuFeatures/GetEnabledCpuFeatures

The `ippInit` function detects the processor type and sets the dispatcher to use the processor-specific code of the Intel® IPP library corresponding to the instruction set capabilities available. If your application does not call the `ippInit` function, initialization of the library to the available instruction set capabilities is performed automatically with the first call of any Intel IPP function from the domain different from `ippCore`.

In some cases like debugging and performance analysis, you may want to get the data on the difference between various processor-specific codes on the same machine. Use the `ippSetCpuFeatures` function for this. This function sets the dispatcher to use the processor-specific code according to the specified set of CPU features. You can obtain features supported by CPU using `ippGetCpuFeatures` and obtain features supported by the currently dispatched Intel IPP code using `ippGetEnabledCpuFeatures`. If you need to enable support of some CPU features without querying the system (without CPUID instruction call), you must set the `ippCPUID_NOCHECK` bit for `ippSetCpuFeatures`, otherwise, only supported by the current CPU features are set.

The `ippInit`, `ippGetCpuFeatures`, `ippGetEnabledCpuFeatures`, and `ippSetCpuFeatures` functions are a part of the `ippCore` library.

GetStatusString

The `ippGetStatusString` function decodes the numeric status return value of Intel® IPP functions and converts them to a human readable text:

```
        status= ippInit();
        if( status != ippStsNoErr ) {
                printf("IppInit() Error:\n");
                printf("%s\n", ippGetStatusString(status) );
                return -1;
        }
```

The `ippGetStatusString` function is a part of the `ippCore` library.

GetLibVersion

Each domain has its own `GetLibVersion` function that returns information about the library layer in use from the dispatcher. The code snippet below demonstrates the usage of the `ippiGetLibVersion` from the image processing domain:

```
const IppLibraryVersion* lib = ippiGetLibVersion();
printf("%s %s %d.%d.%d.%d\n", lib->Name, lib->Version,
lib->major, lib->minor, lib->majorBuild, lib->build);
```

Use this function in combination with `ippInitCpu` to compare the output of different implementations on the same machine.

Malloc/Free

Intel IPP functions provide better performance if they process data with aligned pointers. Intel IPP provides the following functions to ensure that data is aligned appropriately - 16-byte for CPU that does not support Intel® Advanced Vector Extensions (Intel® AVX) instruction set, 32-byte for Intel AVX and Intel® Advanced Vector Extensions 2 (Intel® AVX2), and 64-byte for Intel® Many Integrated Core instructions.

```
void* ippMalloc(int length)
void ippFree(void* ptr)
```

The `ippMalloc` function provides appropriately aligned buffer, and the `ippFree` function frees it.

The signal and image processing libraries provide `ippsMalloc` and `ippiMalloc` functions, respectively, to allocate appropriately aligned buffer that can be freed by the `ippsFree` and `ippiFree` functions.

NOTE

- When using buffers allocated with routines different from Intel IPP, you may get better performance if the starting address is aligned. If the buffer is created without alignment, use the `ippAlignPtr` function.

For more information about the Intel IPP functions see the *Intel® Integrated Performance Primitives for Intel® Architecture Developer Reference* available in Intel® Software Documentation Library.

See Also

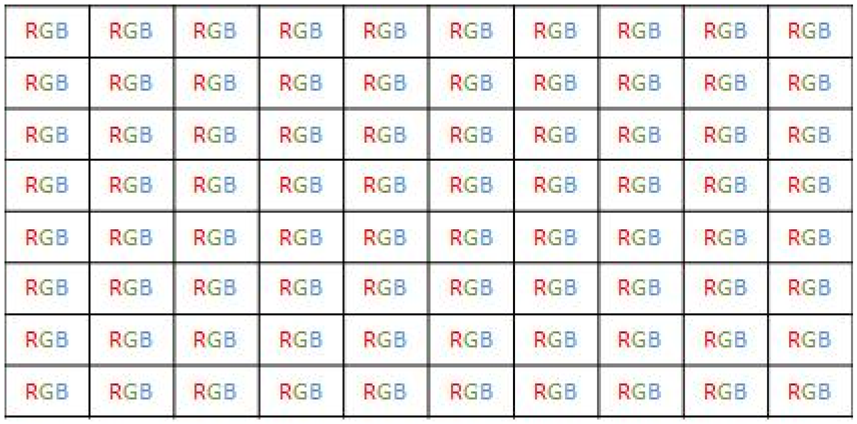
Cache Optimizations
Intel® Software Documentation Library

Channel and Planar Image Data Layouts

Intel® IPP functions operate on two fundamental data layouts: channel and planar.

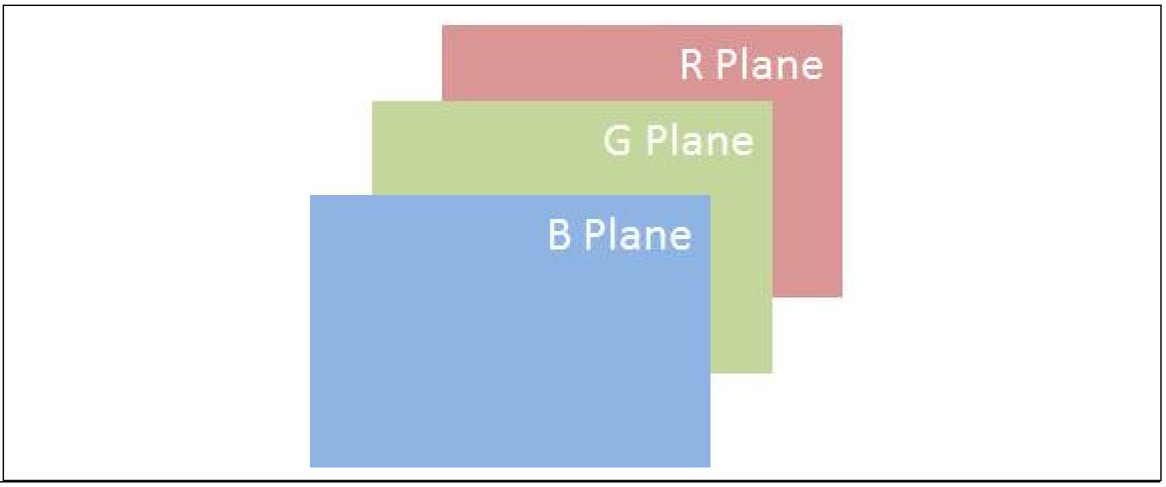
In channel format, all values share the same buffer and all values for the same pixel position are interleaved together. Functions working with channel data have a `_Cn` descriptor, where *n* can take one of the following values: 1, 2, 3, or 4. The figure below shows 24 bit per pixel RGB data, which is represented as `_C3`.

RGB data in `_C3` layout


RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB
RGB RGB RGB RGB RGB RGB RGB RGB RGB RGB

For planar format, there is one value per pixel but potentially several related planes. Functions working with planar data have a `_Pn` descriptor, where *n* can take one of the following values: 1, 2, 3, or 4. The figure below shows 24 bit per pixel RGB data represented as `_P3`.

RGB data in `_P3` layout



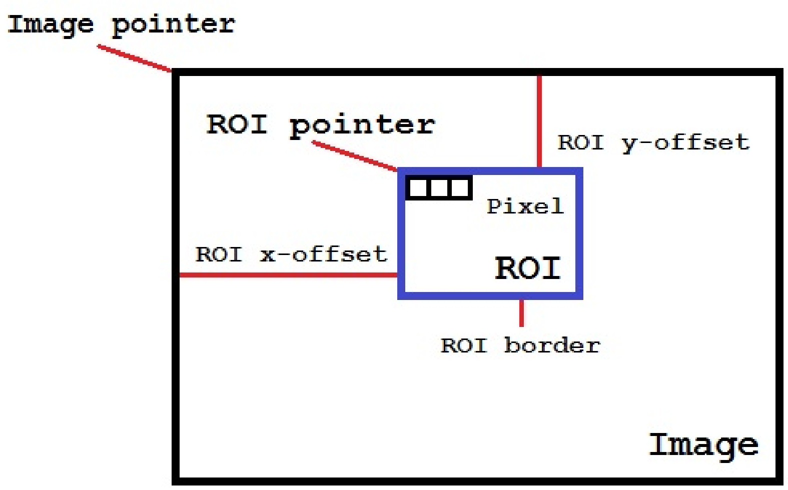
NOTE
For many video and image processing formats planes may have different sizes.

Regions of Interest

Many Intel® IPP image processing functions operate with a region of interest (ROI). These functions include an `R` descriptor in their names.

A ROI can be the full image or a subset. This can simplify thread or cache blocking.

Many functions sample a neighborhood and cannot provide values for an entire image. In this case a ROI must be defined for the subset of the destination image that can be computed.



Managing Memory Allocations

In Intel Integrated Performance Primitives (Intel® IPP) functions, the areas in memory allocated for the source and destination data must not overlap, except for functions that have the descriptor `I` in their name. Only the functions that have the descriptor `I` (see Descriptors) in their name can have the same area in memory allocated for both the source and destination data. Intel IPP does not guarantee correct behavior and results for not-in-place functions that are used in in-place mode.

Depending on the implementation layer and the specific operation parameters, some Intel IPP functions need varying amounts of memory for internal structures and working buffers. To address this, follow the steps below:

1. Compute the size of the required buffer using the *<function base name>*`GetSize` function (some functions have `GetBufSize` or `GetBufferSize` in their name instead of `GetSize`).
2. Set up any buffers needed for initialization. For more information, see the section Setting up Buffers below.
3. Initialize the specification or state structure for the operation using *<function base name>*`Init` function. For more information about the specification and state structures, see the section Specification and State Structures below.
4. Free the buffers need for initialization only (the ones you set up in step 2).
5. Set up working buffers for the main operation. For more information, see the section Setting up Buffers below.
6. Do the main operation.
7. Free the specification or state buffers that you set up in step 3 and the working buffers that you set up in step 5.

If you use several Intel IPP functions with the *pBuffer* parameter (external memory buffer), for better efficiency and performance it is recommended to call all *<function base name>*`GetSize` functions in one single location within your application and allocate only one buffer that has the largest size. This approach ensures optimal use of system memory and all cache levels.

Setting up Buffers

In this document, "setting up a buffer" refers to allocating the required amount of memory and providing a pointer to this memory to the Intel IPP function you are calling. For better performance, you should allocate aligned memory buffers, where the alignment factor depends on the architecture and should be at least 16 bytes for Intel® Streaming SIMD Extensions, 32 bytes for Intel® Advanced Vector Extensions, and 64 bytes for Intel® Advanced Vector Extensions 512 Foundation instruction sets.

To set up aligned memory buffers, it is recommended to use the `ipp`*<domain letter>*`Malloc_`*<IPP data type>* functions; these functions always provide memory buffers with the required alignment.

NOTE Intel IPP functions do not allocate any memory internally. You must manually allocate any memory your Intel IPP functions require at the application level. `ipp`*<domain letter>*`Malloc_`*<IPP data type>* functions do not use any memory manager specific to Intel IPP and are just wrappers for the current C runtime `malloc()` function.

Specification and State Structures

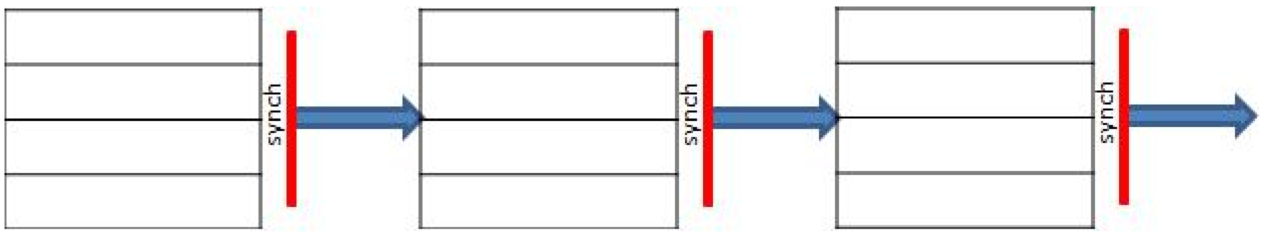
Specification, or spec, structures are `const`; an instance of a specification structure does not change between Intel IPP function calls. Therefore, you can use one instance of a specification structure simultaneously in different application threads for the same operation.

State structures are not `const`; they always contain the state of an intermediate computation stage of an Intel IPP function. Therefore, you can use a single instance of a state structure only for consecutive operations. In the case of a threaded application, each thread must have its own instance of the state structure.

Cache Optimizations

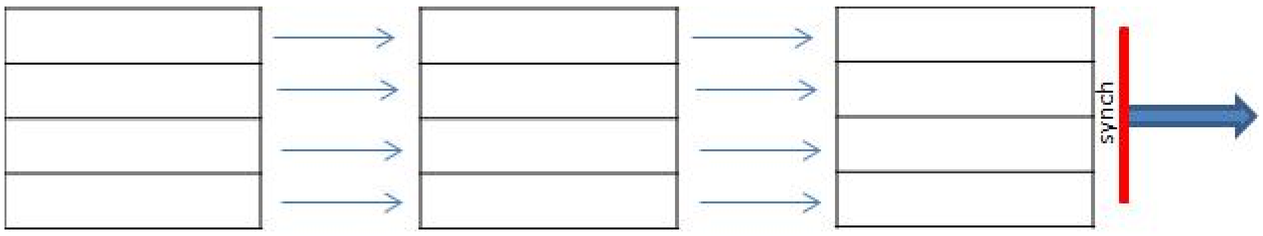
To get better performance, work should be grouped to take advantage of locality in the lowest/fastest level of cache possible. This is the same for threading or cache blocking optimizations.

For example, when operations on each pixels in an image processing pipeline are independent, the entire image is processed before moving to the next step. This may cause many inefficiencies, as shown in a figure below.



In this case cache may contain wrong data, requiring re-reading from memory. If threading is used, the number of synchronization point/barriers is more than the algorithm requires.

You can get better performance after combining steps on local data, as shown in a figure below. In this case each thread or cache-blocking iteration operates with ROIs, not full image.



NOTE
It is recommended to subdivide work into smaller regions considering cache sizes, especially for very large images/buffers.

Programming with Intel® Integrated Performance Primitives in the Microsoft* Visual Studio* IDE

9

This section provides instructions on how to configure your Microsoft* Visual Studio* IDE to link with the Intel® IPP, explains how to access Intel IPP documentation and use IntelliSense* Sense features.

Configuring the Microsoft* Visual Studio* IDE to Link with Intel® IPP

Steps for configuring Microsoft Visual C/C++* development system for linking with Intel® Integrated Performance Primitives (Intel® IPP) depend on whether you installed the C++ Integration(s) in Microsoft Visual Studio* component:

- If you installed the integration component, see Automatically Linking Your Microsoft* Visual Studio* Project with Intel IPP
- If you did not install the integration component or need more control over Intel IPP libraries to link, you can configure the Microsoft Visual Studio* by performing the following steps. Though some versions of the Visual Studio* development system may vary slightly in the menu items mentioned below, the fundamental configuring steps are applicable to all these versions.

  1. In Solution Explorer, right-click your project and click **Properties**.
  2. Select **Configuration Properties**>**VC++ Directories** and set the following from the **Select directories for** drop down menu:

     - **Include Files** menu item, and then type in the directory for the Intel IPP include files (default is `<ipp directory>\include`)
     - **Library Files** menu item, and then type in the directory for the Intel IPP library files (default is `<ipp directory>\lib`)
     - **Executable Files** menu item, and then type in the directory for the Intel IPP executable files (default is `<install_dir>\redist\<arch>\ipp\`)

Using the IntelliSense* Features

Intel IPP supports two Microsoft* Visual Studio IntelliSense* features that support language references: Complete Word and Parameter Info.

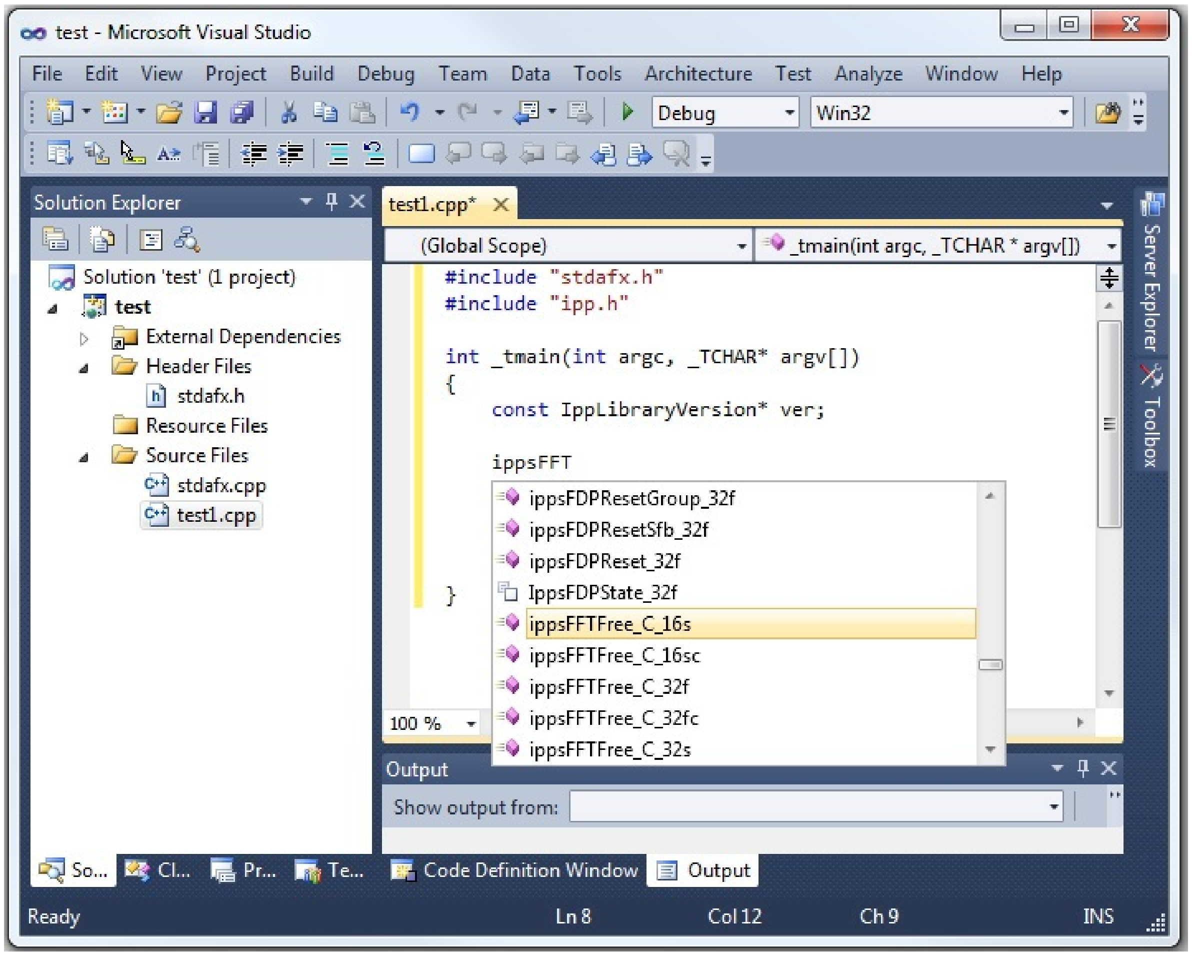
NOTE
Both features require header files. Therefore, to benefit from IntelliSense, make sure the path to the include files is specified in the Visual Studio solution settings. On how to do this, see Configuring the Microsoft Visual Studio* IDE to Link with Intel® IPP.

Complete Word

For a software library, the *Complete Word* feature types or prompts for the rest of the name defined in the header file once you type the first few characters of the name in your code.

Provided your C/C++ code contains the include statement with the appropriate Intel IPP header file, to complete the name of the function or named constant specified in the header file, follow these steps:

1. Type the first few characters of the name (for example, `ippsFFT`).
2. Press **Alt** + **RIGHT ARROW** or **Ctrl** + **SPACEBAR** If you have typed enough characters to eliminate ambiguity in the name, the rest of the name is typed automatically. Otherwise, the pop-up list of the names specified in the header file opens - see the figure below.



```
#include "stdafx.h"
#include "ipp.h"

int _tmain(int argc, _TCHAR* argv[])
{
    const IppLibraryVersion* ver;

    ippsFFT
}
```

```
ippsFDPResetGroup_32f
ippsFDPResetSfb_32f
ippsFDPReset_32f
IppsFDPState_32f
ippsFFTFree_C_16s
ippsFFTFree_C_16sc
ippsFFTFree_C_32f
ippsFFTFree_C_32fc
ippsFFTFree_C_32s
```

3. Select the name from the list, if needed.

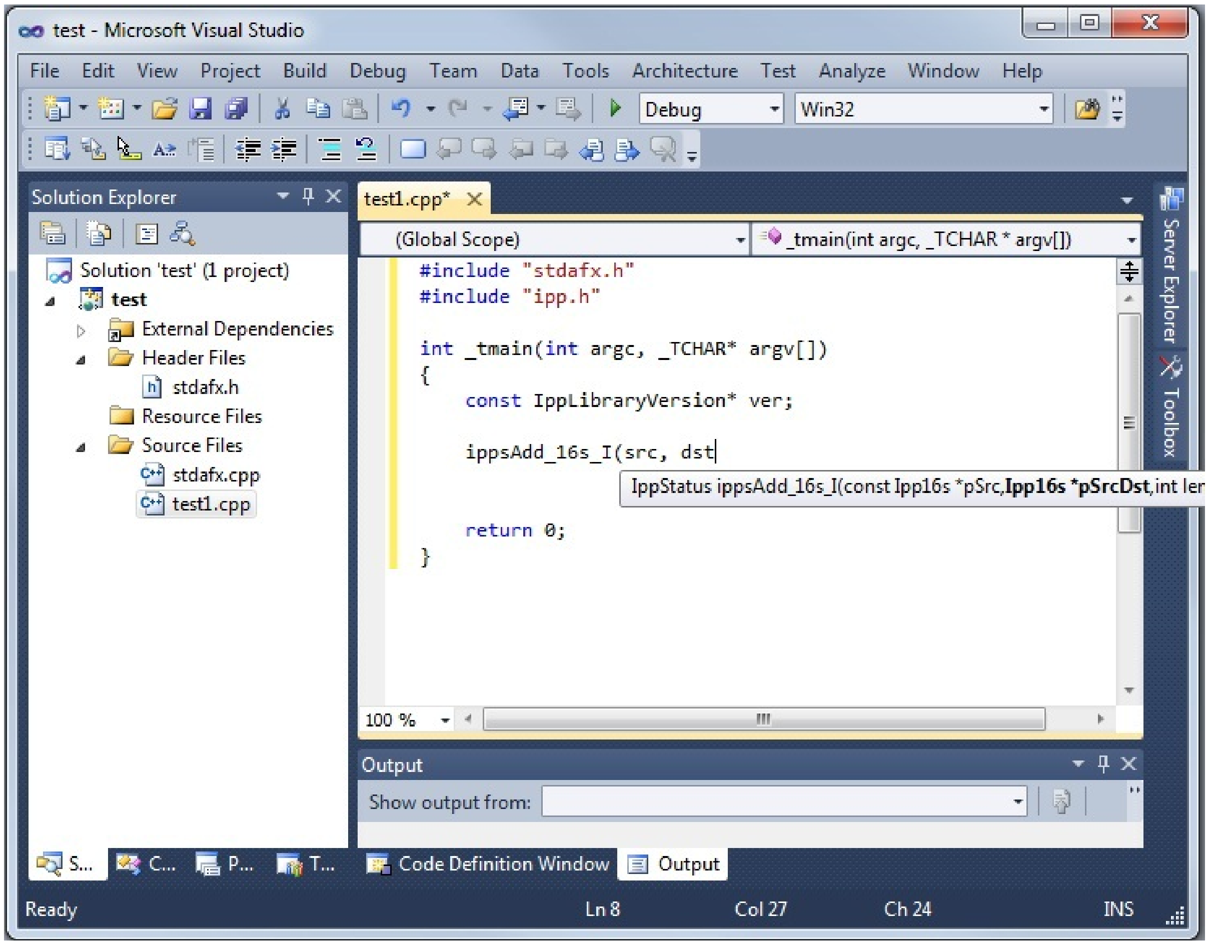
Parameter Info

The *Parameter Info* feature displays the parameter list for a function to give information on the number and types of parameters.

To get the list of parameters of a function specified in the header file, follow these steps:

1. Type the function name
2. Type the opening parenthesis

A tooltip appears with the function API prototype, and the current parameter in the API prototype is highlighted - see the figure below.



```
#include "stdafx.h"
#include "ipp.h"

int _tmain(int argc, _TCHAR* argv[])
{
    const IppLibraryVersion* ver;

    ippsAdd_16s_I(src, dst
                    IppStatus ippsAdd_16s_I(const Ipp16s *pSrc,Ipp16s *pSrcDst,int ler

    return 0;
}
```

See Also

Configuring the Microsoft Visual Studio* IDE to Link with Intel® IPP

Appendix: Performance Test Tool (perfsys) Command Line Options

A

Intel® Integrated Performance Primitives (Intel® IPP) installation includes command-line tools for performance testing in the `<install_dir>/tools/perfsys` directory. There is one perfsys tool for each domain. For example, `ps_ipps` executable measures performance for all Intel IPP signal processing domain functions.

Many factors may affect Intel IPP performance. One of the best way to understand them is to run multiple tests in the specific environment you are targeting for optimization. The purpose of the perfsys tools is to simplify performance experiments and empower developers with useful information to get the best performance from Intel IPP functions.

With the command-line options you can:

- Create a list of functions to test
- Set parameters for each function
- Set image/buffer sizes

To simplify re-running specific tests, you can define the functions and parameters in the initialization file, or enter them directly from the console.

The command-line format is:

```
ps_ipp*.exe [option_1] [option_2] ... [option_n]
```

To invoke the short reference for the command-line options, use `-?` or `-h` commands:

```
ps_ipp*.exe -h
```

The command-line options are divided into several groups by functionality. You can enter options in arbitrary order with at least one space between each option name. Some options (like `-r`, `-R`, `-o`, `-O`) may be entered several times with different file names, and option `-f` may be entered several times with different function patterns. For detailed descriptions of the perfsys command-line options see the following table:

Performance Test Tool Command Line Options

Group	Option	Description
Set optimization layer to test	`-T[`*`cpu-features`*`]`	Call `ippSetCpuFeatures`
Report Configuration	`-A<Timing|Params|Misalign|All>`	Prompt for the parameters before every test from console
	`-o[<`*`file-name`*`>]`	Create `<`*`file-name`*`>.txt` file and write console output to it
	`-O[<`*`file-name`*`>]`	Add console output to the file `<`*`file-name`*`>.txt`
	`-L <`*`ERR|WARN|PARM|INFO|TRACE`*`>`	Set detail level of the console output
	`-r[<`*`file-name`*`>]`	Create `<`*`file-name`*`>.csv` file and write perfsys results to it
	`-R[<`*`file-name`*`>]`	Add test results to the file `<`*`file-name`*`>.csv`
	`-q[<`*`file-name`*`>]`	Create `<`*`file-name`*`>.csv` and write function parameter name lines to it
	`-q+`	Add function parameter name lines to perfsys results table file

Group	Option	Description
	`-Q`	Exit after creation of the function parameter name table
	`-u[<file-name>]`	Create `<file-name>.csv` file and write summary table ('`_sum`' is added to default file name)
	`-U[<file-name>]`	Add summary table to the file `<file-name>.csv` ('`_sum`' is added to default file name)
	`-g[<file-name>]`	Create signal file at the end of the whole testing
	`-l<dir-name>`	Set default directory for output files
	`-k<and|or>`	Compose different keys (-f, -t, -m) by logical operation
	`-F<func-name>`	Start testing from function with `func-name` full name
	`-Y<HIGH/NORMAL>`	Set high or normal process priority (normal is default)
	`-H[ONLY]`	Add 'Interest' column to `.csv` file [and run only hot tests]
	`-N<num-threads>`	Call `ippSetNumThreads(<num-treads>)`
	`-s[-]`	Sort or do not sort functions (sort mode is default)
	`-e`	Enumerate tests and exit
	`-v`	Display the version number of the perfsys and exit
	`-@<file-name>`	Read command-line options for the specified file
Set function parameters	`-d<name>=<value>`	Set perfsys parameter value
Initialization files	`-i[<file-name>]`	Read perfsys parameters from the file `<file-name>.ini`
	`-I[<file-name>]`	Write perfsys parameters to the file `<file-name>.ini` and exit
	`-P`	Read tested function names from the `.ini` file
	`-n<title-name>`	Set default title name for `.ini` file and output files
	`-p<dir-name>`	Set default directory for `.ini` file and input test data files
Select functions	`-f <or-pattern>`	Run tests for functions with `pattern` in their names, case sensitive
	`-f-<not-pattern>`	Do not test functions with `pattern` in their names, case sensitive

Group	Option	Description
	`-f+`*`<and-pattern>`*	Run tests only for functions with `pattern` in their names, case sensitive
	`-f=`*`<eq-pattern>`*	Run tests for functions with `pattern` full name
	`-t[-|+|=]`*`<pattern>`*	Run (do not run) tests with `pattern` in test name
	`-m[-|+|=]`*`<pattern>`*	Run (do not run) tests registered in file with `pattern` in file name
Help	`-h`	Display short help and exit
	`-hh`	Display extended help and exit
	`-h`*`<key>`*	Display extended help for the key and exit

Appendix: Intel® IPP Threading and OpenMP* Support

B

All Intel® Integrated Performance Primitives functions are thread-safe. They support multithreading in both dynamic and static libraries and can be used in multi-threaded applications. However, if an application has its own threading model or if other threaded applications are expected to run at the same time on the system, it is strongly recommended to use non-threaded/single-threaded libraries.

Some Intel IPP functions contain OpenMP* code, which increases performance on multi-processor and multi-core systems. These functions include color conversion, filtering, convolution, cryptography, cross-correlation, matrix computation, square distance, and bit reduction.

To see the list of all threaded APIs, refer to the *ThreadedFunctionsList.txt* file located in the documentation directory of the Intel IPP installation.

Setting Number of Threads

By default, the number of threads for Intel IPP threaded libraries follows the OpenMP* default, which is the number of logical processors visible to the operating system. If the value of the `OMP_NUM_THREADS` environment variable is less than the number of processors, then the number of threads for Intel IPP threaded libraries equals the value of the `OMP_NUM_THREADS` environment variable.

To configure the number of threads used by Intel IPP internally, at the very beginning of an application call the `ippSetNumThreads(n)` function, where `n` is the desired number of threads (1, ...). To disable internal parallelization, call the `ippSetNumThreads(1)` function.

Getting Information on Number of Threads

To find the number of threads created by the Intel IPP, call the `ippGetNumThreads` function.

Using a Shared L2 Cache

Several functions in the signal processing domain are threaded on two threads intended for the Intel(R) Core™ 2 processor family, and make use of the merged L2 cache. These functions (single and double precision `FFT`, `Div`, `and` `Sqrt`) achieve the maximum performance if both two threads are executed on the same die. In this case, the threads work on the same shared L2 cache. For processors with two cores on the die, this condition is satisfied automatically. For processors with more than two cores, set the following OpenMP* environmental variable to avoid performance degradation:

```
KMP_AFFINITY=compact
```

Avoiding Nested Parallelization

Nested parallelization may occur if you use a threaded Intel IPP function in a multithreaded application. Nested parallelization may cause performance degradation because of thread oversubscription.

For applications that use OpenMP threading, nested threading is disabled by default, so this is not an issue.

However, if your application uses threading created by a tool other than OpenMP*, you must disable multi-threading in the threaded Intel IPP function to avoid this issue.

Disabling Multi-threading (Recommended)

The best option to disable multi-threading is to link your application with the Intel® IPP single-threaded (non-threaded) libraries included in the default package and discontinue use of the separately downloaded multi-threaded versions.

You may also call the `ippSetNumThreads` function with parameter `1`, but this method may still incur some OpenMP* overhead.